Events! (Reactivity in urbiscript)

Jean-Christophe Baillie   Akim Demaille   Quentin Hocquet   Matthieu Nottale

Gostai S.A.S., 15, rue Jean-Baptiste Berlier, F-75013 Paris, France

http://www.gostai.com, *first.last*@gostai.com

Abstract— **Urbi SDK is a software platform for the development of portable robotic applications. It features the Urbi UObject C++ middleware, to manage hardware drivers and/or possibly remote software components, and urbiscript, a domain specific programming language to orchestrate them. Reactivity is a key feature of Urbi SDK, embodied in events in urbiscript. This paper presents the support for events in urbiscript.**

Event-based programming is the "native" way in urbiscript to program responses to stimuli — a common need in robotics. It is typically used for "background jobs that monitor some conditions". It is used to program the human-robot interfaces ("do this when the head button is pushed"), the detection exceptional situations (collision avoidance, battery level), the tracking of objects of interest etc. Events are also heavily used in the implementation of Gostai Studio, a GUI for Urbi based on hierarchical state machines [8].

In following example "whenever" an object of interest (a ball) is visible, the head of the robot is moved to track it. The code relies on some of key features of urbiscript: UObjects (`ball`, `camera`, `headPitch`, `headYaw`), concurrency (`&`), and event constructs (**whenever**).

```
whenever (ball.visible)
{ headYaw.val   += camera.xfov * ball.x
& headPitch.val += camera.yfov * ball.y };
```

I. URBI AND URBISCRIPT

A. The Urbi Platform

The Urbi platform [1], including the urbiscript programming language, was designed to be *simple and powerful*. It targets a wide spectrum of users, from children customizing their robots, to researchers who want to focus on complex scientific problems in robotics rather that on idiosyncrasies of some robot's specific Application Program Interface (API).

Urbi relies on modularity to provide portability. Components specific to an architecture (sensors and actuators drivers, . . .) are implemented as *UObjects*, plugged into the Urbi core. UObjects can also wrap pure software components that provide core services, say text-to-speech, object tracking, localization, etc. The C++ UObject architecture is actually a middleware: components can be relocated transparently on remote computers instead of running on the robot itself. This is especially useful for low-end robots whose CPU is too limited to run demanding software components (e.g., face detection, speech recognition, . . .).

The Urbi platform schedules the execution of these UObjects, routes the events from producers to consumers, and so forth. Its core is written in C++. This choice was driven by the availability of compilers for many different types of hardware architecture, and because many robot SDK are in C/C++. It also provides access to very low-level system features (such as coroutines, see below), and allows us to program their support if they lack, in assembler if needed. Specific features of some architectures are also easier to use from C/C++, such as the real-time features of Xenomai. Finally, some architecture *require* C++, such the Aibo SDK.

While the sole Urbi core suffices in many situations, it proved useful to provide a programming language to fine-tune this orchestration.

B. The urbiscript Programming Language

There are already so many programming languages. Why a new one? Why not extending an existing language, or relying on some library extensions?

Domain Specific Languages (DSLs) are gaining audience because they make developers much more productive than when using the library-based approaches. Programming robots requires a complete rethinking of the execution model of traditional programming languages: concurrency is the rule, not the exception, and event-driven programming is the corner stone, not just a nice idiom. These observations alone justify the need for innovative programming languages.

There are already many well-established environments that provide these features, and that can be used to program robots. The world of synchronous languages includes several adequate members, such as Lustre [10] or Esterel [3]. These systems offer soundness and strong guarantees, but at a price: they are very different from the programming languages developers are used to. They are adequate to develop real-time, life-critical systems, but they are too demanding when developing the global behavior of a personal robot. Some general purpose programming languages have been extended also to offer reactive programming: C [4], Caml [12], Haskell [14], etc.

Since the Urbi core is tightly bound to C++, none of these languages are adequate. Binding with low-level languages (such as C++) is a domain in which scripting languages, such as Python [19] or Lua [11], excel. It is not surprising that they are acclaimed interfaces for practical robot programming environments such as ROS [15].

Yet, they do not provide native support for concurrency and reactivity, even if there are existing extensions [6], [17]. When the Urbi project was started (circa 2003), the need for a new language, tailored for programming robotic applications, was felt.

To cope with the resistance to new languages, urbiscript stays in the (syntactic) spirit of some major programming languages: C++, Java, JavaScript etc. As most scripting languages, it is dynamically typed.

It is an Object-Oriented Language (OOL): values are *objects*. Unlike most OOLs, urbiscript is *not* class-based. In *class-based* OOLs (such as C++, Java, C#, Smalltalk, Python, Ruby, . . .), *classes* are templates (molds) that describe the behaviors and members of an object. Classes are *instantiated* to create a value; for instance the `Point` class serves as a template to create values such as `one = (1, 1)`. The object `one` holds the (dynamic) values while the class captures the (static) behavior. Inheritance in class-based languages is between classes.

urbiscript is *prototype-based*, like Self [18], Lisaac [16], Cecil [5], Io [7] and others. In these OOLs, there are no classes. Instantiation is replaced by *cloning*: an object serves as a template for a fresh object, and inheritance relates objects[1].

```
// Create an empty object that derives from Object.
var one = Object.new();
[00000001] Object_0x109fce310
```

An object is composed of a list of *prototypes* (parent objects) and a list of *slots*. A slot maps an identifier to a value (an object).

```
// one is a clone of Object.
one.protos;
[00000002] [Object]

// Add two slots, named x and y.
var one.x = 1;
[00000003] 1
var one.y = 1;
[00000004] 1
// The names of local slots (inherited slots
// are not reported).
one.localSlotNames;
[00000005] ["x", "y"]
```

urbiscript is fully dynamic. Objects can be extended at run-time: prototypes and slots can be added or removed. Functions are *first-class entities*: they are ordinary values that can be assigned to variables, passed as arguments to functions and so forth. urbiscript supports *closures*: functions can capture references from their environment, then later use those references to retrieve or set their content. urbiscript is a functional programming language, functions are values that can be bound by slot like any other object.

```
one;
[00000006] Object_0x109fce310
// The function "asString" is used by the system
// to report values to the user.
function one.asString() { "(%s, %s)" % [x, y] };
one;
[00000007] (1, 1)
```

[1]Lines starting with a timestamp such as [00001451] (1.1451s elapsed since the server was launched) are output by Urbi; the other lines were entered by the user. The system answers with the value of the entered expressions, unless there is none (e.g., `void`). Due to space constraints, the system answers for functions (their definition) is not displayed in this paper.

For a thorough presentation of urbiscript, see [9].

C. Concurrency

Today, any computer runs many programs concurrently. The Operating System is in charge of providing each process with the illusion of a fresh machine for it only, and of scheduling all these concurrent processes. In robots, jobs not only run concurrently, they heavily *collaborate*. Each job is a part of a complex behavior that emerges from their coordination/orchestration.

To support the development of concurrent programs, urbiscript provides specific control flow constructs. In addition to the traditional sequential composition with ';', urbiscript provides the ',' connector, which launches the first statement in background, and immediately proceeds to executing the next statements. Scopes (statements enclosed in curly braces: `{ s1; s2, ... }`), are boundaries: a compound statement "ends" when all its components did. The following example demonstrates these points.

```
// "1s" means one second.  Launch two commands in
// background, using ",".  Execution flow exits the
// scope when they are done.
{ { sleep(2s); echo(2) }, { sleep(1s); echo(1) }, };
echo(3);
[00001451] *** 1
[00002447] *** 2
[00002447] *** 3
```

Other control flow constructs, such as loops, can be executed concurrently. For instance, iterating over a collection comes in several flavors: `for` is sequential while `for&` launches the iterations concurrently (see below).

```
for  (var i : [2,1,0]) {
  echo("%s: start" % i);
  sleep(i);
  echo("%s: done" % i)
};
echo("done");
[00125189] *** 2: start
[00127190] *** 2: done
[00127190] *** 1: start
[00128192] *** 1: done
[00128192] *** 0: start
[00128193] *** 0: done
[00128194] *** done
```

```
for& (var i : [2,1,0]) {
   echo("%s: start" % i);
   sleep(i);
   echo("%s: done" % i)
};
echo("done");
[00105789] *** 2: start
[00105789] *** 1: start
[00105789] *** 0: start
[00105793] *** 0: done
[00106793] *** 1: done
[00107793] *** 2: done
[00107795] *** done
```

The standard library also provides functions that launch new tasks, running in the background. From an implementation point of view, Urbi relies on a library of coroutines [13], not system threads. Job control is provided by *Tags*, whose description fall out of the scope of this paper, see [2].

II. EVENTS

A. Basic Events

Their use goes in three parts. First, an event is needed, a derivative from the `Event` object. This object will serve as an identifier for a set of events, it can be used many times (or not at all).

```
var e = Event.new;
[00007599] Event_0x104bcec50
```

Second, event handlers are needed. They can catch any emission of an event, or filter on the arity of the payload:

```
at (e?)
  echo("e?");
at (e?())
  echo("e?()");
at (e?(var x))
  echo("e?(%s)" % [x]);
at (e?(var x, var y))
  echo("e?(%s, %s)" % [x, y]);
```

Finally, we need to emit an event, possibly with a payload.

```
e!;
[00000033] *** e?
[00000034] *** e?()

e!(12, "foo");
[00000035] *** e?
[00000036] *** e?(12, foo)

e!(12, "foo", 666);
[00000037] *** e?
```

The expression `e!(arg)` is syntactic sugar for `e.emit(arg)`.

B. Semantics

The semantics is simple. There is no guarantee on the order in which the event handlers are run (or rather the order is an implementation detail that is not enforced by our language definition). The handling of events is asynchronous by default, i.e., the control flow that emitted the event may proceed before the event handlers are finished.

```
var f = Event.new;
[00000917] Event_0x107545d90

at (f?(var e)) { echo(e); sleep(0.5s); echo(e); };
f!("handler"); echo("top");
[00000919] *** handler
[00000928] *** top

sleep(1s); // Wait for the second echo.
[00001433] *** handler
```

Event can be send synchronously to override this behavior.

```
f.syncEmit("handler"); echo("top");
[00001929] *** handler
[00002436] *** handler
[00002436] *** top
```

Several handlers can run concurrently.

```
f!("h1"); echo (1); f!("h2"); echo(2);
[00002437] *** h1
[00002442] *** 1
[00002442] *** h2
[00002448] *** 2
sleep(2s);
[00002942] *** h1
[00002954] *** h2
```

Event handlers may raise events; the system does not enforce laws that would prevent endless constructs. We believe that soundness properties such as termination guarantees do not belong to the Urbi system, but to the programmer.

```
var e = Event.new;
at (e?(var p)) { echo(p); e!(-p) };
e!(-1);
[00003919] *** -1
[00003925] *** 1
[00003931] *** -1
[00003936] *** 1
[00003945] *** -1
...
```

From the implementation point of view, a major constraint was the pressure over the CPU. Because in embedded systems (and in particular with low-cost robots) the batteries must be saved to all cost, the implementation aims at the lowest possible foot-print. There is no active wait: in Urbi, if there are no current computations but only event-constructs that monitor external events (incoming network data, UObjects-generated events etc.), then the system consumes no CPU at all.

It is on top of this event-handling layer that urbiscript provides its supports for monitoring arbitrary expressions, see Section III.

C. Filtering

Finally, `at` clauses can filter on the payload.

```
var e = Event.new;
[00000002] Event_0xADDR
var x = 123;
[00000003] 123
at (e?(var x, var y) if x == y)
  echo("e?(%s, %s) with %s == %s" % [x, y, x, y]);
at (e?(x, var y))
  echo("e?(%s, %s)" % [x, y]);

e!(12, 34);

e!(12, 12);
[00000215] *** e?(12, 12) with 12 == 12

e!(x, 34);
[00000218] *** e?(123, 34)

e!(x,  x);
[00000221] *** e?(123, 123) with 123 == 123
[00000221] *** e?(123, 123)
```

urbiscript supports pattern-matching, which can be used to filter the event payload. Special syntactic support is provided for tuples, lists, and dictionaries (and their combinations).

```
var e = Event.new;
[00000206] Event_0x109324980

// Filter on lists.
at (e?([1, var y, "foo"]))
  echo("[1, %s, \"foo\"]" % y);
e!(1, 2, "foo");
e!([1, 2, "foo"]);
[00000312] *** [1, 2, "foo"]
```

III. EXPRESSIONS AS EVENTS

The `at` blocks can also be used to monitor arbitrary expressions:

```
var x = 1;
[00000001] 1
```

```
// The absence of question mark indicates we're
// monitoring an expression, not an event.
at (x <= 0)
  echo("x is negative")
onleave
  echo("x isn't negative anymore");

x = -1;
[00000002] -1
[00000003] *** x is negative
x = -2;
[00000004] -2
x =  1;
[00000005] 1
[00000006] *** x isn't negative anymore
```

This is actually syntactic sugar on top of `makeEvent(exp)`. It turns any Boolean expression into an event that triggers when the expression evaluates to true, and stops when it becomes false again.

A. Durations

Gostai developed some robotic behaviors using event constructs, for instance to trigger some reaction when a ball is visible, or ceases to be. Because the object detection is not perfect, the robot sometimes appeared to behave erratically. This is easy to solve using some hysteresis mechanism, but the result is cluttered uses of event constructs. The same happens when implementing behaviors such as "do this when the head of the robot is patted for two seconds".

To address this issue urbiscript provides support to monitor conditions that are sustained for some specified amount of time: the handler may require an event to be sustained for a given amount of time before being "accepted" (**at** (*exp* ~ *duration*)). The optional **onleave** clause is run when the condition is invalidated.

```
var x = 0;
[00001855] 0
at (0 < x ~ 1s)
  echo("%1.1f: at" % [time() - t0])
onleave
  echo("%1.1f: onleave" % [time() - t0]);

t0 = time();
[00001862] 1.19469

// x is 1 for 1 second.
x = 1; sleep(1s);
[00001863] 1
[00002865] *** 1.0: at

// The event was triggered, it is not rerun.
sleep(1s);

// Reset.
x = 0;
[00003869] 0
[00003869] *** 2.0: onleave
// x is not positive long enough.
x = 1; sleep(0.9s); x = 0; sleep(0.5s);
[00003874] 1
[00004777] 0
// x is positive long enough.
x = 1; sleep(0.5s); x = 2; sleep(0.5s);
[00005281] 1
[00005783] 2
[00006283] *** 4.4: at
```

B. Main Usage: Arbitrary Event Monitoring

The main interest of this feature is to be able to monitor in an event-driven way objects without requiring any special facility from their part. Consider for instance a "car" object, that has a "fuel" slot indicating the remaining fuel level. With a classical approach, if we want to trigger an alert when we're running out of gas, the car would have to provide an adequate event, and trigger it each time the fuel is altered and is under a given threshold.

```
class Car
{
  // The urbiscript constructor.
  function init() {
    var this.fuel = 1;
    // We must manually create a specific
    // event to signal low fuel level.
    var this.lowFuel = Event.new;
  };

  // We must never update fuel directly, but use
  // this setter, otherwise we might end up not
  // sending the lowFuel event.
  function updateFuel(var v) {
    var previous = fuel;
    fuel = v;
    var threshold = 0.05;
    // We must emit the event only if we just
    // passed below the threshold.
    if (previous >= threshold && v < threshold)
      lowFuel!;
  };
};
[00000735] Car

var car = Car.new;
[00000802] Car_0x109f322e0

at (car.lowFuel?)
  echo("Warning, running out of gas!");
```

This method has several cons. It forces the `Car` implementer to write a lot of boiler plate code, and to anticipate all its user needs and provide the adequate events. For instance here we cannot monitor other values of the fuel level. A more generic interface would be to provide an event that triggers each time the fuel level changes.

```
class Car
{
  function init() {
    var this.fuel = 1;
    // We must manually create a specific
    // event to signal low fuel level.
    var this.fuelChanged = Event.new;
  };

  // We must never update fuel directly, but use
  // this setter.
  function updateFuel(var v) {
    var previous = v;
    fuel = v;
    // For convenience, the event carries the
    // previous and current values as payload.
    fuelChanged!(previous, fuel);
  };
};
[00000735] Car
var car = Car.new;
[00000783] Car_0x102ed0990
```

```
// Like before, give a warning on low fuel level.
at (car.fuelChanged?(var prev, var cur)
    if 0.05 < prev && cur <= 0.05)
  echo("Warning, running out of gas!");
// Also, check we do not overfill the tank.
at (car.fuelChanged?(var prev, var cur)
    if prev < 0.95 && 0.95 <= cur)
  stopFillingGas();
```

This approach allows the user to monitor the fuel level for any condition. However, the `Car` implementer still has to provide and trigger manually events for changing variables. The `Car` user can monitor the fuel level in any way, but at the expense of verbose and complex **at** constructs.

With the use of **at** on arbitrary expressions, we can get rid of all these problems: the `Car` implementer simply uses the fuel slot naturally by directly affecting it values. Since nothing special has to be done, he cannot forget to provide any monitoring event. The user can monitor any condition on the fuel slot, in a more concise way.

```
class Car
{
  function init() { var this.fuel = 0.5 }
  // No wrapper needed, directly update "fuel".
};
[00000735] Car
var car = Car.new;
[00000737] Car_0x102db0d80

// Car has a fuel level slot, a float.
car.fuel;
[00000000] 0.5

// Although Car wasn't designed to emit fuel
// events, we can react to its variations.
at (car.fuel < 0.05)
  echo("Warning, almost out of gas!");
at (0.95 < car.fuel)
  stopFillingGas();
```

C. General Purpose Usage: Eased Flow Control

Another interesting usage of this feature is simplified control flow in some situation. We can for instance separate a stop condition from an algorithm with an **at**: the algorithm is as soon as the condition is verified. We can then perform the algorithm without worrying about checking our exit cases.

The following function is an example of synchronous flow control. It searches a value in a list, sorted (dichotomy) or not (linear search).

```
function find(var list, var val, var sorted)
{
  var begin = 0;
  var end = list.size;

  // First, factor all our stop cases with ats.

  // When we find the element, return its index.
  at (list[begin] == val)
    return begin;
  // When we exhaust our search space, fail.
  at (begin == end)
    return -1;

  // Now only the traversals remain.
  if (sorted)
    // Perform a dichotomy on sorted vectors.
```

```
    loop
    {
      var middle = ((begin + end) / 2).floor;
      if (list[middle] < val)
        begin = middle + 1
      else
        end = middle
    }
  else
    // Linear search on non-sorted vectors.
    loop
      begin++;
};
```

D. Implementation

The main concern is efficiency. A trivial implementation would be simple busy-looping: checking at each instant (at the end of every scheduler cycle) whether the condition is true, and triggering the event if needed. This would work, but is of course highly time and space (and energy!) expensive.

We could check the condition only at a given frequency, but then we could miss some events, if the condition becomes true and then false again between two checks. Moreover, this would imply a potential delay between the realization of the condition and the triggering of the **at**, which can be acceptable for reading some robotic sensor, but not in our previous vector-search algorithm. This would anyway not be optimal, since we would probably end up doing too many checks.

Our implementation uses a "push" model rather than a "poll" one: instead of checking the condition arbitrarily to see whether it changed, it's the modification of any value that might alter the expression that will trigger the reevaluation of the expression, and potential triggering of the at block. This is optimal, since the condition is reevaluated only if there's any chance it changed: if nothing related to your **at** happens, it won't consume any CPU cycle.

To achieve this, when an **at** is first declared, the condition is evaluated, and any mutable data encountered during this evaluation is hooked for modification. Then every time some of this data is altered, we reevaluate the condition and trigger the **at** block if needed. When the condition is reevaluated, the hooked variables list is also flushed and rebuilt since it might differ, if we take another branch of a conditional statement for instance.

```
var global = false;
[00000619] false

function test(var foo)
  { if (global) foo.a == 1 else foo.a == foo.b };

var foo = Object.new;
[00000620] Object_0x1091939d0
var foo.a = 1;
[00000621] 1
var foo.b = 2;
[00000621] 2

at ({echo("evaluate"); test(foo)})
  echo("True!")
onleave
  echo("False!");
[00000001] *** evaluate
```

```
// Hooked variables here are: global, foo,
// foo.a and foo.b.

// Altering foo.b triggers an evaluation,
// but does not alter the result or the hooked
// variables list.
foo.b = 3;
[00000002] *** evaluate
[00000002] 3
// Altering global makes the condition true.
global = true;
[00000003] *** evaluate
[00000004] *** True!
[00000002] true

// The list of hooked variables is now: global,
// foo, foo.a.

// Altering foo.b does not trigger a reevaluation,
// since it no longer participates.
foo.b = 2;
[00000006] 2
```

IV. FUTURE WORK

A. Handlers Synchronicity

As of today, urbiscript allows the emission of an event to declare whether the event handlers must be performed immediately (synchronous), or "later" (asynchronous). But experience shows that some patterns are better treated when it is on the handler side that synchronicity is chosen.

B. Congestion Control

As reported earlier, the (asynchronous) execution of event handlers may result in several instances of event handlers being run concurrently. Depending on the application this might be a feature, or a nuisance.

There are several ways to process events whose handling is long. Similarly to what the clock-directed loops in urbiscript already do, bodies could be forbidden to overlap by waiting for the previous handler to finish, they might need to be preempted by newer event emissions, we could event manage some priority scheme, with features that would allow to drop obsolete events.

C. Urbi

The Urbi platform, as a middleware, already provides support for emitting events at the C++ level, independently of the urbiscript language. But the reception part is ongoing work.

CONCLUSION

We have presented event-driven programming in urbiscript, the programming language of choice to orchestrate UObjects (Urbi components). Together with the primitive support for concurrency, it is the corner stone for the Urbi paradigm for the development of robotic applications. Although it bears resemblance to reactive programming languages, the urbiscript language is made to make experimentation easier by offering more opportunities for the addition of feature *a posteriori*. Expression-based events are one typical example: they allow to monitor events in a non-intrusive way, without requiring that hooks were previously deployed in the monitored objects.

REFERENCES

[1] Jean-Christophe Baillie. URBI: Towards a universal robotic low-level programming language. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'05)*, pages 820–825, 2005.

[2] Jean-Christophe Baillie, Akim Demaille, Quentin Hocquet, and Matthieu Nottale. Tag: Job control in urbiscript. In Noury Bouraqadi, editor, *5th National Conference on Control Architecture of Robots*, May 2010.

[3] Gérard Berry and Georges Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, 1992.

[4] Frédéric Boussinot. Reactive C: an extension of C to program reactive systems. *Softw. Pract. Exper.*, 21(4):401–428, 1991.

[5] Craig Chambers. The Cecil language specification and rationale: Version 3.2. February 2004.

[6] A. Lúcia de Moura, N. Rodriguez, and R. Ierusalimschy. Coroutines in Lua. *Journal of Universal Computer Science*, 10(7):910–925, July 2004. │http://www.jucs.org/jucs

[7] Steve Dekorte. Io: a small programming language. In *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'05)*, pages 166–167, New York, NY, USA, 2005. ACM.

[8] Gostai. Gostai Studio. http://www.gostai.com/products/studio/gostai 2010.

[9] Gostai. Urbi SDK 2.2 manual. http://www.gostai.com/downloads/urbi-sdk/2.2/doc/urbi-sdk.h 2010.

[10] Nicolas Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[11] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. Lua — an extensible extension language. *Softw. Pract. Exper.*, 26(6):635–652, 1996.

[12] Louis Mandel and Marc Pouzet. ReactiveML, a reactive extension to ML. In *Proceedings of 7th ACM SIGPLAN International conference on Principles and Practice of Declarative Programming (PPDP'05)*, Lisbon, Portugal, July 2005.

[13] Ana Lúcia De Moura and Roberto Ierusalimschy. Revisiting coroutines. *ACM Trans. Program. Lang. Syst.*, 31(2):1–31, 2009.

[14] John Peterson, Greg Hager, and Paul Hudak. A language for declarative robotic programming. In *International Conference on Robotics and Automation*, 1999.

[15] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*, 2009.

[16] Benoît Sonntag and Dominique Colnet. Lisaac: The power of simplicity at work for operating system. In *In Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific*, pages 45–52. Australian Computer Society, Inc, 2002.

[17] Christian Tismer. Continuations and Stackless Python or "how to change a paradigm of an existing program". Technical report, Virtual Photonics GmbH, 2000.

[18] David Ungar and Randall B. Smith. SELF: The power of simplicity. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87)*, volume 22, pages 227–242, New York, NY, 1987. ACM Press.

[19] Guido van Rossum. Python tutorial. Technical report, CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands, The Netherlands, 1995.